Universal Content Launcher

A series of tutorials detailing the creation of a content launcher in Microsoft Visual C#

Tutorial Five

Completing the Basic Launcher

Contents

Introduction 3
Tutorial Requirements 3
Part One: Re-defining what we have 4
Part Two: Editing the content record in the Content XML file 11
Part Three: Deleting the content record in the Content XML file 14
Part Four: Adding a context menu 16
Part Five: Adding an XML file for settings 19

Introduction

In tutorial four, we managed to get our launcher to launch content of all the available different types. We also managed to create a new form for adding content to our XML file from UCL itself. Now it's time to build on these things and finish our basic launcher, and to do this we only need two more things; the ability to edit content that is already in our XML file, and the ability to remove it from the XML file.

Tutorial Requirements

These tutorials are sequential, so if you haven't already been through the previous tutorials, then please download, and work through them before continuing with this one. Tutorial one also outlines the software you will need for creating UCL and where you can download it for free.

Part One: Re-defining what we have

Well, this is really the final part of “Phase One” of our content launcher. So, we have a form to add content, and we need a form to edit it. At first it seems trivial to just dive right in, copy our Add New Content form, paste it as an Edit Content form and get on with it.

But wait, we’re above all that! Why create another form, use more resources, and more memory, when we can just re-use the one we have. All we need to do is pass the form a value or trigger that tells it whether we are adding content, or editing it.

Cast your mind back to the last tutorial, and you will remember that we already had to do this, to get the content list to refresh correctly from the manage content form, remember? We passed the calling forms object to our Add Content form. Well now, all we have to do is tell it to Add or Edit as well.

First lets rename our form, it’s no longer going to be an Add New Content form, but a Manage Content Form, so Let’s call it just that. Remember, we don’t want to break all of our code on page one again, so let’s do it properley and rename it in the form properties.

- First bring up your form design for the Add New Content form and click on the form so that it’s properties are displayed.
- Change the name to ManageContent
- Notice that the form name does not change in the solution explorer!
- Click on AddNewContent.cs in the solution explorer
- In the properties change the FileName to ManageContent.cs
- Hit enter, and now everything should be as expected.

Try UCL, just to make sure we didn’t break anything – It should work fine, and your Add New Content form should still display ok.

Now we need to pass the mode that we want the form to open in (Add or Edit) but let’s think about that for a moment.

We could pass a boolean flag (Edit) which is true for edit or false for Add.

We could pass a text string that says either “Add” or “Edit”

But let’s think some more, in order to edit something, then we need to pass some indication of what it is we’re going to edit, so we have to pass the content record for the object to edit, to allow us to display it’s old settings and let the user overtype them, and save their new settings.

As we’ve got to pass a content object anyway, then we could use that as an edit flag, so if it is null, then we know we are adding a record, and if it contains a valid content object, then we display it for editing.

That sounds clever – let's do that:

- Add a new menu item in your UCLMain form menu under the Add New Content line, type in &Edit Content. Note that we have already assigned the Alt-E hotkey to the menu item &Edit, but these menu names seem wrong now, so change the &Edit menu item to &Content, so we end up with Add New Content and Edit Content under the Content menu. That makes more sense!
- Double click on Edit Content to create the new method
- Copy and paste the code out of `addNewContentToolStripMenuItem_Click` into the `editContentToolStripMenuItem_Click` method.
- Now go to your `addNewContentToolStripMenuItem` method again.
- Change this line:

```
ManageContent frmUCLAddContent = new ManageContent(this);
```

- To This:

```
ManageContent frmUCLAddContent = new ManageContent(this, null);
```

- Now go back to your new `editContentToolStripMenuItem` method.
- Change this line:

```
ManageContent frmUCLAddContent = new ManageContent(this);
```

- To This:

```
ManageContent frmUCLAddContent = new ManageContent(this,
tContent);
```

That should do it! But now we need to create a new `tContent` object that holds the content we are trying to edit, we can re-use some of the code we've already written, so if I told you that my `editContentToolStripMenuItem` method now looks like this:

```
        private void editContentToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            UCLContent tContent =
GetContentFromName(treeView1.SelectedNode.Text);
            if (tContent != null)
            {
                ManageContent frmUCLEditContent = new ManageContent(this,
tContent);
                frmUCLEditContent.Show();
            }
            else
            {
                MessageBox.Show("Please select the content you wish to
edit!");
            }
        }
```

Does that make sense to you? Note that I have called my form instance object `frmUCLEditContent` rather than `frmUCLAddContent`, as this makes the code more readable, though it doesn't really alter the functionality at all.

Now we need to change our initialise method in the Manage Content form to handle the new content object.

To do this we first need a new variable in our Manage Content form that can be shared across the methods in it, come on, you know how to do this now:

```
        private UCLContent tContent;
```

I put that right below this line:

```
        private UCLMain frmUCLMain;
```

But you already knew that right?

Now we can modify the `ManageContent` method to store the `PassedContent` object into our new `tContent` object, like so:

```
       public ManageContent(UCLMain OpenedBy, UCLContent PassedContent)
        {
            InitializeComponent();
            frmUCLMain = OpenedBy;
            tContent = PassedContent;
        }
```

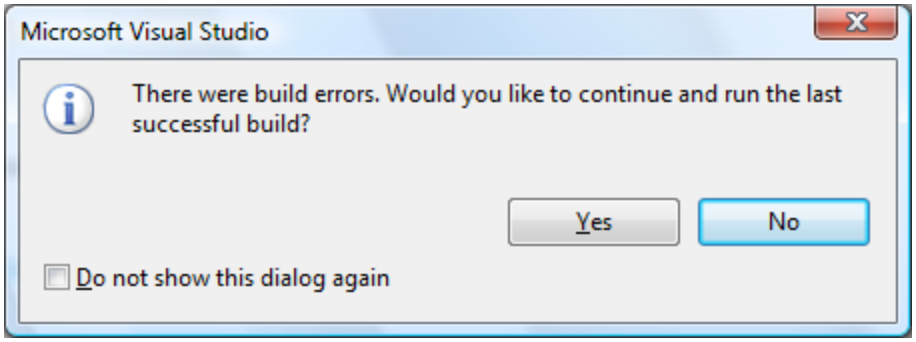
You should still be with me – this stuff is way under what we're capable of now!

Now we have the object, we can do neat things in the forms load event, like so:

```
        private void UCLAddContent_Load(object sender, EventArgs e)
        {

            foreach (string tType in
System.Enum.GetNames(typeof(UCLContent.UCLType)))
            {
                comboType.Items.Add(tType);
            }

            if (tContent == null)
            {
                this.Text = "Add New Content";
            }
            else
            {
                this.Text = "Edit " + tContent.Name;
            }

        }
```

You see?

This is Great! Let's run it! Run UCL now, to see how all this works...



Huh? Here's the error you should have got:

Inconsistent accessibility: parameter type 'UCL.UCLContent' is less accessible than method 'UCL.ManageContent.ManageContent(UCL.UCLMain, UCL.UCLContent)'

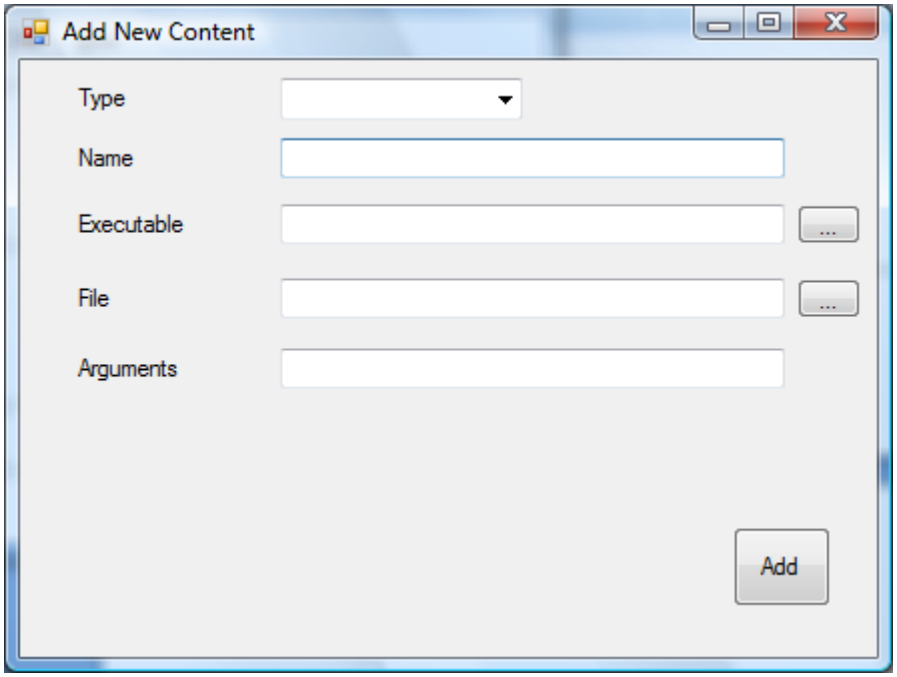
Nasty, what the hell does that mean? Well, read it – even if you don't understand it, common sense will prevail. It's telling you that something (UCLContent) isn't as accessible as something else (UCLMain). You can ignore all the "UCL." In the sentence, they're just detailing what our project namespace is.

If you look at the class definition for UCLMain, you will notice it is defined as public. Look at the one for UCLContent, not public, is it?

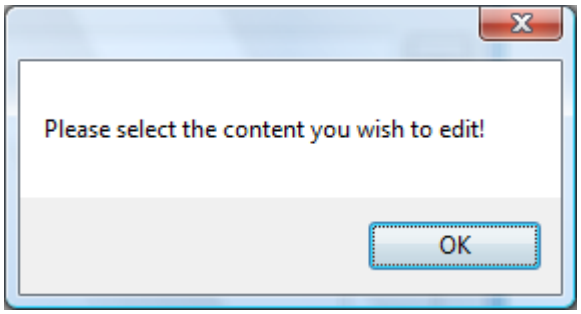
Change it like so:

```
    public class UCLContent
```

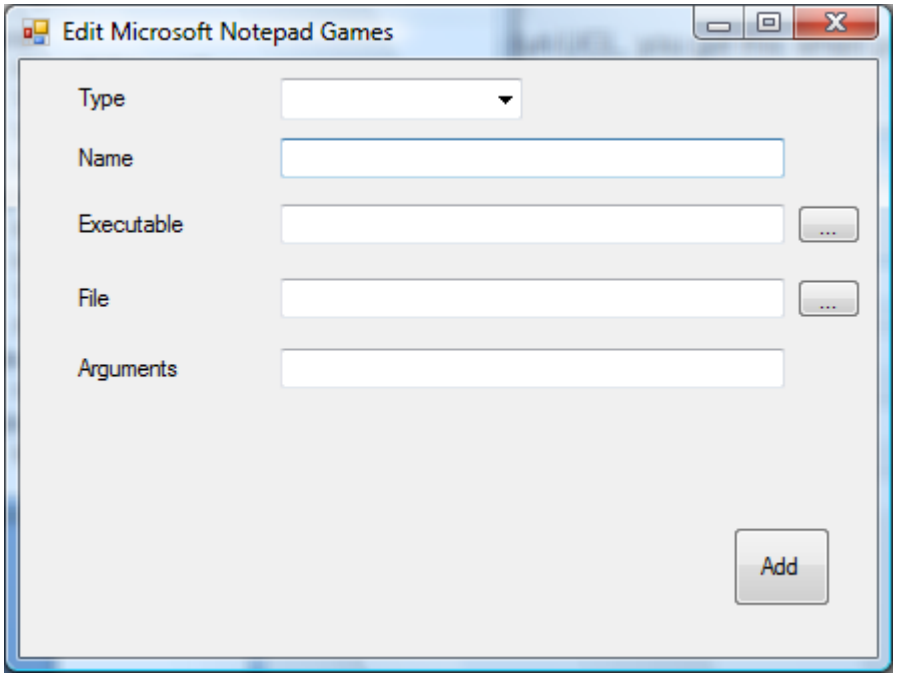
And now, when you run UCL, you get this when you click on the “Add New Content” menu item:



And this; when you choose the “Edit Content” menu item:



Click on some content first you fool, and then you will get this (note the title matches the content name you clicked on):



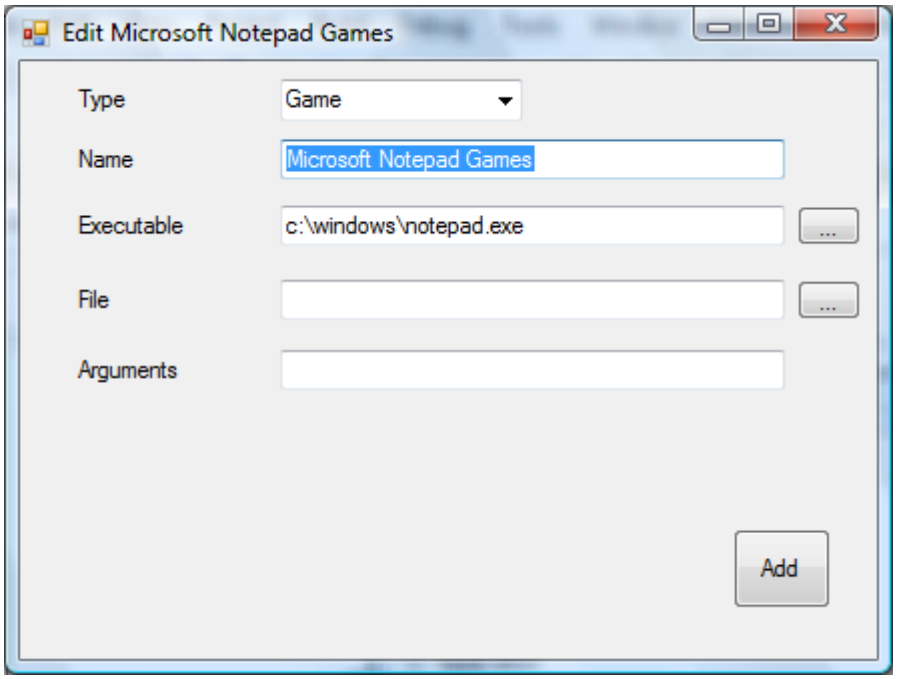
Well, we changed the title, but we need to populate all those fields with the current values for the content when we're editing a record. We can read it from the XML file right? But why bother? We already have it in memory, and we already have the object in our form (`tContent`)!

All we need to do is this, in the `UCLAddContent_Load` method:

```
        private void UCLAddContent_Load(object sender, EventArgs e)
        {

            foreach (string tType in
System.Enum.GetNames(typeof(UCLContent.UCLType)))
            {
                comboType.Items.Add(tType);
            }

            if (tContent == null)
            {
                this.Text = "Add New Content";
            }
            else
            {
                string tUCLType = Enum.GetName(typeof(UCLContent.UCLType),
(tContent.Type));
                this.comboType.SelectedText = tUCLType;
                this.textName.Text = tContent.Name;
                this.textExecutable.Text = Path.Combine(tContent.ExePath,
tContent.ExeName);
                if (tContent.FileName != null)
                {
                    this.textFile.Text = Path.Combine(tContent.FilePath,
tContent.FileName);
                }
                if (tContent.Arguments != null)
```

```
            {
                this.textArguments.Text = tContent.Arguments;
            }

            this.Text = "Edit " + tContent.Name;
        }

    }
```

No explanation needed – I think!
Now run UCL:



Wow, this tutorial is shooting by, now all we have to do is amend the relevant record in the XML file when the “Add” button is clicked. Huh? “Add” Button, we’re editing!, so when we do that we need to change the button to say something like “Save”, not “Add”.

Come on now, that’s easy! Add this line at the end of the code above:

```
this.buttonAddContent.Text = "Save";
```

After this line:

```
this.Text = "Edit " + tContent.Name;
```

Job done!

Part Two: Editing the content record in the Content XML file

Ok, now it's just the XML file left to edit. How the heck do you update a record in an XML file?

Well, if you're thinking you just need to search for it, and then create a new element to overwrite it, then you would be spot on. Because we have already written most of this in previous methods, let me show you the full code for the "Save" button that works for either the Add or Edit scenarios:

```
        private void buttonAddContent_Click(object sender, EventArgs e)
        {
            if (comboType.Text == "")
            {
                MessageBox.Show("Please select a Type for this Content!");
                return;
            }

            if (textName.Text == "")
            {
                MessageBox.Show("Please select a Name for this Content!");
                return;
            }

            if (textExecutable.Text == "")
            {
                MessageBox.Show("Please select the Executable File for this
Content!");
                return;
            }

            XmlDocument doc = new XmlDocument();
            doc.Load(Path.Combine(Application.StartupPath,
"UCLContent.xml"));

            XmlElement elementType = doc.CreateElement(comboType.Text);

            XmlElement elementNode = doc.CreateElement("Name");
            elementNode.InnerText = textName.Text;
            elementType.AppendChild(elementNode);

            elementNode = doc.CreateElement("ExeName");
            elementNode.InnerText = Path.GetFileName(textExecutable.Text);
            elementType.AppendChild(elementNode);

            elementNode = doc.CreateElement("ExePath");
            elementNode.InnerText =
Path.GetDirectoryName(textExecutable.Text);
            elementType.AppendChild(elementNode);

            if (textFile.Text != "")
            {
                elementNode = doc.CreateElement("FileName");
                elementNode.InnerText = Path.GetFileName(textFile.Text);
                elementType.AppendChild(elementNode);

                elementNode = doc.CreateElement("FilePath");
                elementNode.InnerText =
Path.GetDirectoryName(textFile.Text);
```

```
                elementType.AppendChild(elementNode);

            }

            if (textArguments.Text != "")
            {
                elementNode = doc.CreateElement("Arguments");
                elementNode.InnerText = textArguments.Text;
                elementType.AppendChild(elementNode);

            }

            if (tContent == null)
            {

                doc.DocumentElement.AppendChild(elementType);

            }
            else
            {
                XmlElement root = doc.DocumentElement;

                XmlNode oldElementType = doc.SelectSingleNode("Content/" +
comboType.Text + "[Name='" + tContent.Name + "']");

                root.ReplaceChild(elementType, oldElementType);

            }

            doc.Save(Path.Combine(Application.StartupPath,
"UCLContent.xml"));

            this.Dispose();
            MessageBox.Show("The Content was succesfully saved!");

            frmUCLMain.LoadContentList();

        }
```

Make sense?

Notice how we have reused most of the code to add new content to our XML file. We only branch depending on the Add or Edit mode at the very end of the code:

```
            if (tContent == null)
            {

                doc.DocumentElement.AppendChild(elementType);

            }
            else
            {
                XmlElement root = doc.DocumentElement;

                XmlNode oldElementType = doc.SelectSingleNode("Content/" +
comboType.Text + "[Name='" + tContent.Name + "']");

                root.ReplaceChild(elementType, oldElementType);

            }
```

Here, we either do the `AppendChild` method to add a new element, or if we're editing an element, then we search for it by using the `SelectSingleNode` method and passing it the name of the content we are editing, to search for.

We can then simply replace the whole element in the XML file with the `ReplaceChild` method. Simple!

Part Three: Deleting the content record in the Content XML file

Deleting content is even easier!

- Add a new &Delete Content menu item on the UCLMain form
- Double click on it to generate the method:

```
        private void deleteContentToolStripMenuItem_Click(object sender,
EventArgs e)
        {

        }
```

Now we only need to search for the content, like in the edit method, ask the user if they really want to delete it, and do the business:

```
       private void deleteContentToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            UCLContent tContent =
GetContentFromName(treeView1.SelectedNode.Text);
            if (tContent != null)
            {
                DialogResult result;

                result = MessageBox.Show(this, "Are you sure you wish to
Delete " + tContent.Name + "?", "Confirm Deletion",
MessageBoxButtons.YesNo);

                if (result == DialogResult.Yes)
                {

                    XmlDocument doc = new XmlDocument();
                    doc.Load(Path.Combine(Application.StartupPath,
"UCLContent.xml"));

                    XmlElement root = doc.DocumentElement;

                    XmlNode oldElementType =
doc.SelectSingleNode("Content/" + Enum.GetName(typeof(UCLContent.UCLType),
(tContent.Type)) + "[Name='" + tContent.Name + "']");

                    root.RemoveChild(oldElementType);

                    doc.Save(Path.Combine(Application.StartupPath,
"UCLContent.xml"));

                    LoadContentList();
                }

            }
            else
            {
                MessageBox.Show("Please select the content you wish to
Delete!");
            }

        }
```

It should all be familiar, even the extended usage of `MessageBox` and the `RemoveChild` element method are just extensions of things we have already covered.

This C# stuff is just getting easier and easier! So easy in fact, that we've done all we want to do for a basic launcher. We've got a bit of space in the tutorial to add in some bells and whistles, and I've noticed that I keep right clicking on that content list to edit, or delete items, so let's add in a context menu for our treeview control.

Part Four: Adding a context menu

Right then, first we need to add a context menu control to our UCLMain form, like we did when we created the original UCL menu, so go to your toolbox and drag the ContextMenuStrip onto your UCLMain form design.

This should now appear under our original Menu, and you add items to it in exactly the same way, so let's add:

- Add New Content
- Edit Content
- Delete Content

All one below the other, to our context menu. Note that we don't really need hotkeys, as this is all done by doing a right-click with our mouse, and then left clicking on the appropriate menu item.

Now we need to tell UCL where the context menu is used, so click on your treeview control and scroll down the properties until you see ContextMenuStrip. This is a dropdown list, and you should see your new context menu (ContextMenuStrip1) in the list, so select that.

What we have done here, is tell UCL that whenever we right click anywhere on the treeview control, to display our context menu. Run UCL and you will see it an action, though it does nothing exciting at this point in time!

Now let's customize our context menu depending on whether or not we actually click on a valid node in the treeview (we should show the Edit and Delete context menu items) or a category type (we should only show the Add New Content context menu item)

```
        private void treeView1_MouseClick(object sender, MouseEventArgs e)
        {
            UCLContent tContent =
GetContentFromName(treeView1.SelectedNode.Text);
            if (tContent == null)
            {
                addNewContentToolStripMenuItem1.Visible = true;
                editContentToolStripMenuItem1.Visible = false;
                deleteContentToolStripMenuItem1.Visible = false;
            }
            else
            {
                addNewContentToolStripMenuItem1.Visible = false;
                editContentToolStripMenuItem1.Visible = true;
                deleteContentToolStripMenuItem1.Visible = true;
            }
        }
```

Easy? It should be. The only thing to note is the names of the menu objects, these were named automatically by UCL as we created them, and you can, of course, find the name of an object by looking in its properties. We are simply showing the various context menu items, or hiding them depending on where we click in the treeview.

Let's try it! Run UCL, and you will see that our context menu always displays just "Add New Content"! The reason for this is apparent if you look closely, you will notice that when you right click on a node in the treeview list, it doesn't select it!

To fix this, we need a bit of mouse magic! Click on your treeview control in the UCLMain form design and look at its events (not properties)

Double click on the MouseDown event to create a new method, and enter the following:

```
private void treeView1_MouseDown(object sender, MouseEventArgs e)
 {
     TreeNode tSelectedNode = treeView1.GetNodeAt(e.X, e.Y);
     if (tSelectedNode != null)
     {
         treeView1.SelectedNode = tSelectedNode;
     }
     else
     {
         treeView1.SelectedNode = null;
     }
 }
```

Can you see what we are doing here? We are setting a Treeview node object `tSelectedNode` depending on where the mouse is positioned. As we are in a mouse event, then the variable `e` contains `MouseEventArgs` data. So `e.X` is the Mouse X position, and `e.Y` is the Mouse Y position.

We then use `treeView1.GetNodeAt(e.X, e.Y)` to reveal which node the mouse is over, and set `tSelectedNode` node accordingly, which if not null, we use to actually select the node in the treeview control itself.

We set the treeview control to null if we are not right clicking on a valid node. Now when you run UCL, right clicking on things works properly! Except for one thing...

Click at the very bottom of the treeview content list, where there is no content and just space. You will always see the last context menu that was displayed – BUG!

We need to debug. Put a breakpoint in the MouseDown method – that seems like a reasonable place to start, and run UCL, first clicking on some valid content, and pressing F5 in the debugger to continue, and then click in the blank space.

Well, stepping through the MouseDown method returns the correct result – it does this:

```
treeView1.SelectedNode = null;
```

So, we need to look at our code in the `treeView1_MouseClick` method. Set a breakpoint in this method, and debug your code again.

See the problem? The method doesn't run when we click in the space of the treeview control! So the easiest thing to do is disable the context menu all together if no node is selected, as we already know that this code (the `treeView1_MouseDown` method) does run.

We can do this by modifying the method as shown below:

```
        private void treeView1_MouseDown(object sender, MouseEventArgs e)
        {
            TreeNode tSelectedNode = treeView1.GetNodeAt(e.X, e.Y);
            if (tSelectedNode != null)
            {
                treeView1.SelectedNode = tSelectedNode;
            }
            else
            {
                treeView1.SelectedNode = null;
                addNewContentToolStripMenuItem1.Visible = false;
                editContentToolStripMenuItem1.Visible = false;
                deleteContentToolStripMenuItem1.Visible = false;
            }
        }
```

Now try it – Perfect!

Now then, the last thing to do is the easiest of all! We need to create the same code for clicking on the Add, Edit and Delete context menu items that we did for the main menu items. This seems like a waste of effort to type all that out again, and indeed there is no need to do so, as we can simply “pass through” the method.

Double click on each of the context menu items to make their corresponding methods. Then for each one, copy in the method for the main menu click methods, as shown below:

```
        private void addNewContentToolStripMenuItem1_Click(object sender,
EventArgs e)
        {
            addNewContentToolStripMenuItem_Click(sender, e);
        }

        private void editContentToolStripMenuItem1_Click(object sender,
EventArgs e)
        {
            editContentToolStripMenuItem_Click(sender, e);
        }

        private void deleteContentToolStripMenuItem1_Click(object sender,
EventArgs e)
        {
            deleteContentToolStripMenuItem_Click(sender, e);
        }
```

All we’ve done is “pass through” the context menu method to the main menu method that already has the code written for it. As the parameters for all of these methods contain the same variables (`object sender, EventArgs e`) then we simply need to pass these as well. Simple!

Now try UCL, and amaze yourself at just how clever you are!

Part Five: Adding an XML file for settings

So, here we are at the end of our journey. We have a basic functional launcher, waiting to have some eye candy and juicy features bolted on. These will come in the shape of advanced tutorials for things like:

- Using the windows API for Icon Management
- Interfacing with a Steam Emulator

Before we go however, there is still time to add in some user settings.

Create a new XML file (you know how) and call it UCLConfig.xml. Don't forget to set the "Copy to Output Directory" property to "Copy if Newer".

Fill your new XML file as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<UCL>
  <Settings>
      <AutoExpand>True</AutoExpand>
  </Settings>
</UCL>
```

We are going to enable the user to toggle the automatic treeview expansion, so UCL can open either with the treeview collapsed, or expanded.

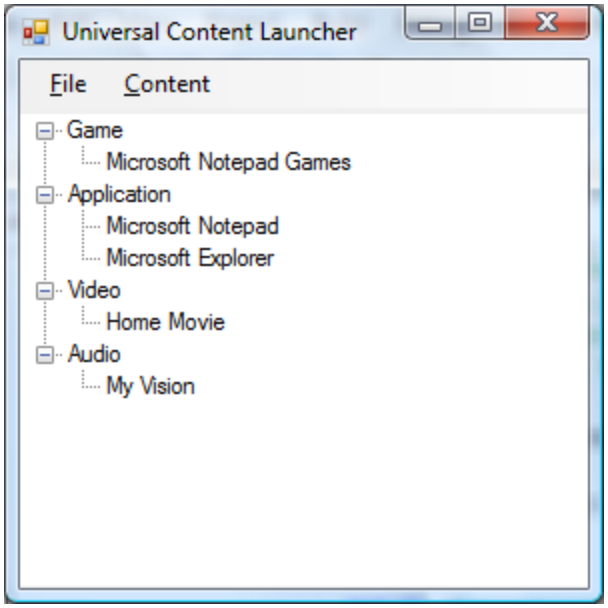
We already know how to do this, but let's make it more useable by writing a function that will return the value for any setting that we query:

```
        private string GetUCLSetting(string Section, string Setting)
        {

            XmlDocument doc = new XmlDocument();
            doc.Load(Path.Combine(Application.StartupPath,
"UCLConfig.xml"));

            XmlNode SettingElement = doc.SelectSingleNode("UCL/" + Section
+ "/" + Setting );

            if (SettingElement != null)
            {
                return SettingElement.InnerText;
            }
            else
            {
                return null;
            }

        }
```

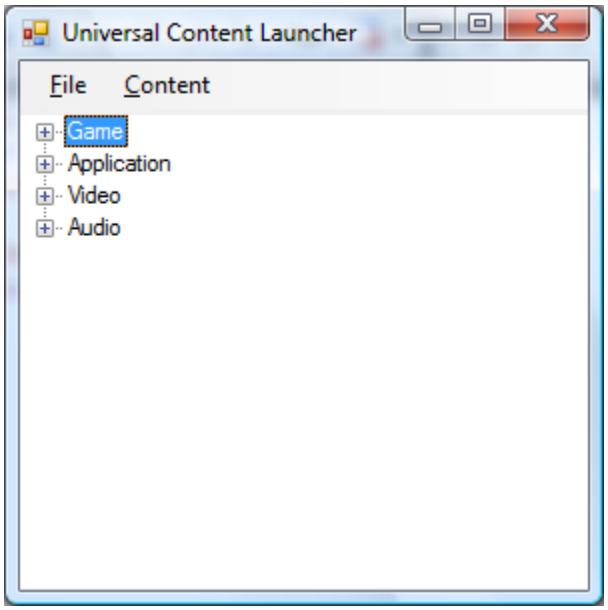
The only thing new there is the `InnerText` method that merely returns the text stored in the element. Now we can call our new method at the place where we were previously just expanding the treeview, in the `LoadContentList` method:

```
if (GetUCLSetting("Settings", "AutoExpand") == "True")
{
    treeView1.ExpandAll();
}
```

Now run UCL and you will see this (probably with differing content)



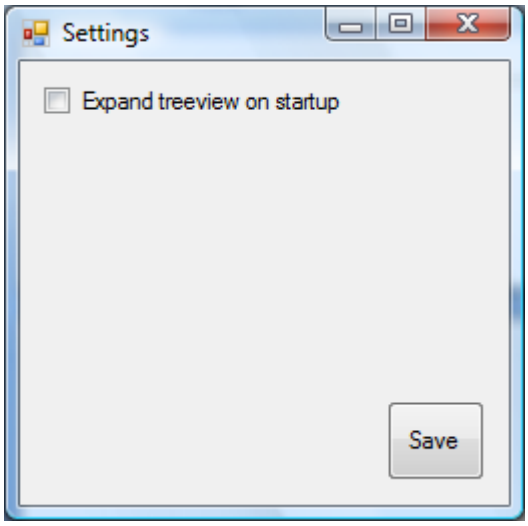
Now change the value for `AutoExpand` in your UCLConfig XML file to `False`. You will see this:



Nice! Now all we need is a settings form.

- Add a New menu item called C&ustomise to your menu and add the menu item &Settings below it.
- Now create a new form and call it UCLSettings and change its Text property to Settings.

- Design it like so, by dropping on a check box and giving it the Text “Expand treeview on startup”, then Name it checkboxAutoExpand, before adding a “Save” button to the bottom of the form:



Don’t forget to check your Anchor settings on your controls. This form really is bland, but it’s a start, and as UCL grows, we will probably be revisiting this quite a lot!

Now we need two more things:

- Populate the settings from the XML file when the form loads
- Save the settings to the XML file when the “Save” button is clicked.

Since I’m nice, here is the `Load` method for our new form, it should make perfect sense to you now:

```
private void UCLSettings_Load(object sender, EventArgs e)
{

    if (GetUCLSetting("Settings", "AutoExpand") == "True")
    {
        this.checkboxAutoExpand.Checked = true;
    }
    else
    {
        this.checkboxAutoExpand.Checked = false;
    }

}
```

Run UCL, and you’ll discover how nice I really am!

> The name 'GetUCLSetting' does not exist in the current context

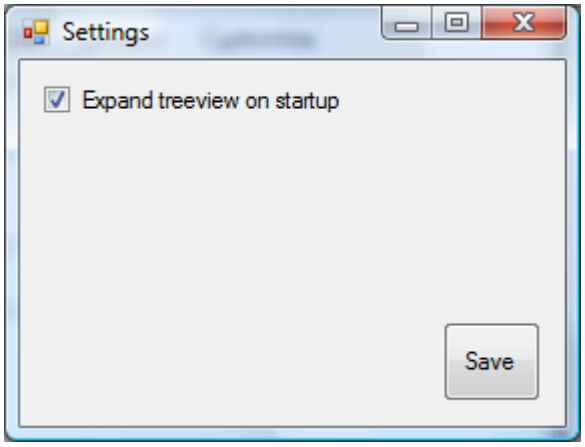
Of course it doesn’t because we created the `GetUCLSetting` method in UCLMain as a private method, we are no longer in UCLMain! But, there is good news, because the `GetUCLSetting` method is not specific to an instance of the `UCLMain` object, then we can declare it as a static method in the `UCLMain` class like so:

```
        public static string GetUCLSetting(string Section, string Setting)
```

Now all we need to do is change our call to it in the `UCLSettings_Load` method to this:

```
        if (UCLMain.GetUCLSetting("Settings", "AutoExpand") == "True")
```

And now it works! To test it, check that your XML file has the `AutoExpand` element set to `True`, run UCL again, and the box should now be ticked:



Now write in the save button code as follows; but don't forget to add these lines first:

```
using System.Xml;
using System.IO;
```

Here's the method:

```
        private void buttonSave_Click(object sender, EventArgs e)
        {
            XmlDocument doc = new XmlDocument();
            doc.Load(Path.Combine(Application.StartupPath,
"UCLConfig.xml"));
            XmlNode root = doc.SelectSingleNode("UCL/Settings");

            XmlElement elementType = doc.CreateElement("AutoExpand");

            if (this.checkboxAutoExpand.Checked == true)
            {
                elementType.InnerText = "True";
            }
            else
            {
                elementType.InnerText = "False";
            }

            XmlNode oldElementType =
doc.SelectSingleNode("UCL/Settings/AutoExpand");

            root.ReplaceChild(elementType, oldElementType);

            doc.Save(Path.Combine(Application.StartupPath,
"UCLConfig.xml"));
```

```
            this.Dispose();
            MessageBox.Show("The settings were succesfully saved!");

        }
```

Now, a problem with this is that when we alter and save the AutoExpand setting, the UCLMain form does not refresh to reflect the change. Because the treeview is in the instance of the form, then we do still need to pass the UCLMain form object to our settings form after all.

Remember how to do this? Change the Settings form as follows; note that the `frmUCLMain` object is declared as a shared object amongst the forms methods:

```
        private UCLMain frmUCLMain;

        public UCLSettings(UCLMain OpenedBy)
        {
            InitializeComponent();
            frmUCLMain = OpenedBy;
        }
```

Add this line between the `Dispose` and `MessageBox` statements at the end of the `buttonSave_Click` method above:

```
            frmUCLMain.SetTreeviewMode();
```

Now go to your UCLMain form, add a new method like so:

```
        public void SetTreeviewMode()
        {

            if (GetUCLSetting("Settings", "AutoExpand") == "True")
            {
                treeView1.ExpandAll();
            }
            else
            {
                treeView1.CollapseAll();
            }
        }
```

As we have this new method, then modify your `LoadContentList` method in UCLMain to also use it, by changing this:

```
            if (GetUCLSetting("Settings", "AutoExpand") == "True")
            {
                treeView1.ExpandAll();
            }
```

To this:

```
            SetTreeviewMode();
```

Now Run UCL and try it!

It's worth pointing out that this is a truly simple settings dialogue that only processes text input and output to our XML file, but it is a good place to start and even more importantly, it works!

That’s it! One basic launcher, delivered as per your order.

Take a rest and even spend some time trying to enhance the launcher yourself, by adding more settings and features. You will need the breather, for the next tutorial will start the “Advanced” phase of our project, and we will be looking into spicing up our display with some shiny new icons.